Rapid prototyping of complete systems, the case study of a smart parking

Laurent-Frédéric Ducreux, Claire Guyon-Gardeux, Maxime Louvel, François Pacull, Thior Safietou Raby, Maria Isabel Vergara-Gallego

Rapid prototyping of complete systems, the case study of a smart parking

Laurent-Frédéric Ducreux, Claire Guyon-Gardeux, Maxime Louvel,
François Pacull, Safietou Raby Thior, Maria Isabel Vergara-Gallego
Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA, LETI, MINATEC Campus, F-38054 Grenoble, France
Email: firstname.lastname@cea.fr

***Abstract*—This paper details how LINC a coordination middleware, can fasten the development of prototypes that integrate several equipment. A case study of rapid prototyping is presented. It illustrates how a smart parking prototype has been built from several independent and autonomous equipment, coming from different vendors. This has been achieved by parallel development thanks to the resource based approach offered by LINC. This paper also describes how LINC helps building rich user interfaces quickly and easily.**

Keywords–Coordination, Prototyping, User interface

I. Introduction

Embedded systems have been part of our daily life for decades. Most of the industrial or consumer products embed multiple processors, sensors and actuators. The next step is the opportunity to build new products, systems, and usages, combining together several of these products. In such innovative and quickly evolving context it is necessary to build prototypes to test, validate, and challenge new products usages or solutions. This paper considers prototypes including several equipment, possibly from different vendors. Such prototypes can be used to exhibit demonstrations in a trade fair, to convince investors of the viability of an idea or to verify the adequacy of the solutions for early adopters.

To succeed in today's highly competitive market, a prototype should be more than a few items wired together with a command line interface. Targeted prototypes are real-life demonstrations of new products or technologies with a high end-user experience. Moreover, a prototype should be included in its targeted environment (e.g. a house, a building or a parking). The development of prototypes is vital during the validation stage of new products as they may help to:

1) *Reduce the time to market*: by testing and validating product functionality;
2) *Improve and extend functionality*: by opening the way to new scenarios and usages of the product;
3) *Create alliances*: by integrating products coming from different vendors;
4) *Validate segments of clients*: by pruning or expending in a early stage the set of segments of clients.

Time and cost being critical, adequate tools are required to fasten prototypes development. Such tools must help to develop in parallel several parts of the prototype and ease their integration in the last stage. They must also be flexible enough to allow up to last minute changes. For instance, when replacing a piece of hardware by another (e.g. cheaper, more powerful or with better availability). Changes in the user interface should also be facilitated to better take into account feedback of early adopters. In addition, tools must simplify the simulation of the equipment still under development. Indeed, very often there is a co-design and co-development of a set of devices and the global application that integrates all of them. Simulating these devices at little cost reduces the length of the critical path. Finally, fast development of rich user interfaces are also needed because they leverage opportunities offered by the prototype and allow to consider several alternatives.

In this paper, we show how we can use LINC [1] to answer these challenges. LINC is a resource-based middleware which integrates a rule engine. The resource approach provides an abstraction layer to ease the integration of heterogeneous components. The rule engine provides a coordination layer that permits defining interactions between components, through the use of coordination rules.

We illustrate the ability of LINC to deal with rapid prototyping through the case study of a "smart parking". The challenge was twofold: first, the time frame of 3 months to put everything in place and, second, the fact that some of the equipment were themselves prototypes still under development. Several scenarios demonstrating different usages of the smart parking have been put in place: for (i) the car driver, (ii) the people in charge of parking monitoring and control, and (iii) the parking administrator for charging station and parking slot usage optimisation.

This paper is organised as follows. First, section II presents an overview of LINC. Then, section III presents the intrinsic properties of LINC that address the challenges of rapid prototyping. Section IV describes the case study, the components and equipment to be integrated, and the scenarios put in place. Then, section V details the implementation of the prototype and section VI describes how user interfaces have been built. Section VII presents related work on rapid prototyping. Finally, section VIII concludes the paper.

II. Overview of LINC

LINC is described in detail in [1], however this section recalls some information to make the paper self contained. LINC provides a uniform abstraction layer to encapsulate software and hardware components. This abstraction layer relies on the associative memory paradigm implemented as a distributed set of bags containing resources (tuples). Inspired by Linda [2], bags are accessed only through three operations:

- `rd()`: takes a partially instantiated tuple as input parameter and returns from the bag a stream of fully instantiated tuples matching the given input pattern;
- `put()`: takes a fully instantiated tuple as input parameter and inserts it in the bag;

- `get()`: takes a fully instantiated tuple as input parameter, verifies if a matching resource exists in the bag and consumes it in an atomic way.

Bags are grouped within objects according to application logic. For instance, all the bags used to manage a set of devices of the same type or technology are grouped in an object.

A. Coordination rules

The three operations `rd()`, `get()` and `put()` are used within production rules [3]. A production rule is composed of a precondition phase and a performance phase.

Precondition phase: The precondition phase is a sequence of `rd()` operations which detect or wait for the presence of resources in several given bags. The resources are, for instance, values from sensors, external events, or results of service calls. In the precondition phase:

- the output fields of a `rd()` operation can be used to define input fields of subsequent `rd()` operations;
- a `rd()` is blocked until at least one resource corresponding to the input pattern is available.

Performance phase: The performance phase of a LINC rule combines the three `rd()`, `get()` and `put()` operations to respectively verify that some resources (e.g. the one(s) found in the precondition phase) are present, consume some resources, and insert new resources.

In this phase, the operations are embedded in one or multiple *distributed transactions* [4], executed in sequence. Each transaction contains a set of operations that are performed in an atomic manner. Hence, we can guarantee that actions belonging to the same transaction, are either all executed or none. This ensures properties such as:

- Some conditions responsible for firing the rule (precondition) are still valid at the time of the performance phase completion;
- All the involved bags are effectively accessible. For instance, for a bag encapsulating a remote service we can determine if such service can be actually accessed.

These properties ensure that the set of required objects, bags, and resources, are actually available "at the same time".

B. Bag abstraction

The bag abstraction maps software and hardware components:

database: A possible encapsulation is to associate a bag to each table of the database. These bags can be automatically generated from the database meta-data. The `rd()` and `put()` operations correspond to the `read` and `write` on the database. The `rd()` operations in the precondition part develops an inference tree, like a SQL request.

remote service: To encapsulate a remote service, partially instantiated tuples are passed to a `rd()` with input parameters (e.g.: $("4511'45.96", "542'35.81", town)$ to get the town associated to GPS coordinates). The remote service is queried within the proprietary format and communication protocol. The result is combined with the input parameters to define a fully instantiated tuple (i.e. $("4511'45.96", "542'35.81", "Grenoble")$) which is returned by the `rd()` operation.

event system: An event can be represented as a tuple with the following fields $(id, topic, timestamp, payload)$. To encapsulate an event system, `rd()` and `put()` are mapped to `publish` and `subscribe` operations on the system.

sensor: To encapsulate sensors, three bags can be used: `Sensors` to store $(id, value)$, `Type` to associate sensor `type` to its `id` (i.e. $(id, type)$) and `Location` to manage the location (i.e. $(id, location)$). The first two bags are usually filled by the sensor network gateway driver and the last one is managed at the application layer when the binding is done.

actuator: To operate on an actuator, a resource is put in the bag associated to it. The resource contains the `id` of the actuator, the `command` to apply and possible `parameters`. When the resource is added, the command is sent to the actuator through the driver encapsulating the protocol.

user interface: User interfaces follow a Model-View-Controller [5] approach implemented above the bag paradigm. Inputs or gestures from users are inserted in the bag `MVCControl` to indicate to the system what has been done on graphical elements (e.g. clicked, mouse over). This information is used to trigger further actions on the whole system. In return, feedback is given as modifications to be applied to the graphical elements. This is done through resources inserted in the bag `MVCStatus`. A resource has three fields defining a graphical element, one of its attributes, and the new value. Finally, inserting a resource in the bag `MVCRefresh` triggers the update of a given part of the user interface. This is described in more detail in the following section.

C. Frameworks

A framework is a group of objects targeting a domain. For instance, the PUTUTU [6] framework integrates more than 20 technologies for sensor/actuator networks. A framework offers ready to use objects and/or template objects to speed up development. We have defined several frameworks dedicated to UI, user notifications (e-mail, SMS), voice recognition and synthesis, light management, and so on.

D. Tools

LINC offers tools to help development and debugging. Firstly, a web based monitor allows to remotely introspect and modify the content of the bags. An analysis tool is provided to observe the rules execution, the amount of resources read, added or removed in bags. Finally, it is possible to restart an application by forcing the rules to be re-executed in the same order than a previous execution. This is useful for debugging purposes, as described in section III-E.

III. LINC and Rapid Prototyping

A LINC application can be decomposed into three layers:

- the external and legacy world encapsulated as bags;
- the user interface, through web browser based interaction following a Model-View-Controller approach;
- the application logic, where the application state is kept as resources in the bags, and the state transitions are driven by coordination rules bridging the external world and the users.

LINC provides loose coupling through the resource/bag based approach. This is particularly useful for rapid prototyping as now explained.

A. Parallel development

The bag abstraction provides decoupling in both time and space between the producer and the consumer. Hence to develop several parts in parallel, it is only required to define the resources that will be produced and consumed. This is true for hardware (e.g. sensors or actuators) encapsulated within the bag abstraction. But this is also true at the interface level (e.g. what graphical element have been clicked) and at the application level (e.g. events or state of a process). Indeed, one team can develop the graphical interface, while another can write the code to encapsulate a sensor or an existing product, and another can work on the application logic. The teams only have to agree on the data format that will be exchanged.

B. Simulation

With the resource approach, it is very easy to simulate other parts of the system. Indeed the resources of interest simply need to be added in a bag. This can be done manually or via a coordination rule. This allows to validate the system even if every single part or device is still under design. This is critical for rapid prototyping, where part of the components can be available or stable only in the last stage of the development.

C. Re-usability

The resource approach, coupled with the coordination rules, also increases re-usability which is vital for rapid prototyping. In fact, the decoupling between the modelling (using resources) and the usage (through rules) permits the design of objects that do not depend on the way they are used. This facilitates the reuse of objects in different applications/contexts. For instance, the ModBus object, from the PUTUTU framework, has been reused in 3 demonstrators from different domains: home automation, lift energy management, and smart parking.

D. User Interfaces

LINC provides four types of objects that can be combined to define very flexible and dynamic interfaces.

The first one, `Layout` defines the global interface as a set of frames. Frames may overlap and accept transparency effects. The association of the frame and its content (URL) is stored in a bag. Thus, it is possible to dynamically change an interface: frames can be moved and their content changed.

The second object `MVC` (Model-View-Controller) is used, on the one hand, to capture the interactions from the users and, on the other hand, to reflect the current state of the application. This state is the result of the rules execution regardless to what are the causes: user interactions, sensors detection, results of a complex algorithm, events, etc.

The third object `Chart` is used to directly render as charts information contained in bags. Once the bag `name`, the `type` of chart, and the different `parameters` inherent to the type of chart are known, the object returns a web page containing a chart (in SVG) corresponding to the resources contained in the bag. This allows to manage both a static chart from data collected in the past or a dynamic chart following the changes of one or several values. We consider not only the classical charts (such as plots, histograms, and pies) but also gauges and progression bars when a single value is followed.

Finally, the last object `dynamicSVG` is used to animate 2D SVG graphical objects into a SVG scene, composed of three main bags. The `Sprite` bag manages the graphical representation of a sprite. The `Location` bag manages the place of the sprite within the global scene. The `Animation` bag associates a `sprite`, a `path` to follow, a `duration` and a `type of movement` (accelerate, slow down, ...). Collisions with other graphical objects are detected and can be used to trigger further actions through rules.

Thus, an elaborated user interface is constructed with one `Layout` object, and possibly objects of type `MVC`, `Chart` and `dynamicSVG` associated to the frames. In addition, any other HTML element coming from an external source may be included in a frame. It is possible to refresh the frames independently according to what is really required. This decreases the network traffic and the CPU usage of the host running the browser. This allows to target very small devices (e.g. a kindle paper-white from amazon is a perfect device for home automation remote user interface).

As the status is not contained in the user interface but at the objects level, the same data can be used for different interfaces and/or for different sessions of the same interface on different devices. These sessions can be synchronised or not depending on the need. Indeed, triggering the refresh of a browser session is as simple as inserting the appropriate resources in the `MVCRefresh` bag. This is used when collaboration/sharing among geographically distributed users is required by the application. Another usage, is to propose the user to choose among several alternative representation of a same interface.

The development of an interface may be dispatched to several people with different expertise. The purely graphic part can be done by a designer using standard tools such as Adobe Illustrator or Inkscape without any knowledge of LINC. The result is plain SVG drawing. The instrumentation of this SVG is done by another person for which the knowledge required about LINC is just the generic API needed to insert a resource in a bag when an action on a graphical entity is done. This can be very easily done in an SVG editor such as Inkscape. `Chart` type interfaces are directly integrated without any code to write. `dynamicSVG` requires more knowledge of LINC since the animations are coded as a set of LINC rules defining how the sprites move according to contextual conditions.

E. Tools

The tools complete the environment by providing facilities to remove or add resources when the application is running. This allows debugging or verification of sub part of the application without needing to restart everything. The rule analysis tools allow to spot very easily rules that can be optimised. Finally, the controlled re-execution of an application forces a non-deterministic application to behave in the same way as a previous crashed execution. Debugging is easier because otherwise one would have to replay the application several times hoping to recreate the crash condition.

IV. Case study description

The case study consists of a "smart parking" that has been developed in the context of the IRT Nanoelec [7] within the PULSE program. This is a french initiative, whose role is to accelerate collaboration between industry and academia. The "smart parking" prototype has been inaugurated during an event called "day of the sustainable mobility" organised at CEA premises in Grenoble.

The target was to develop a prototype integrating products provided by several industrial partners, such as *Bouygues* and *Schneider electric*, and off-the-shelve products. The equipment from partners consist of their last prototypes still under development and improvement.

A. Smart parking description

1) Parking spot solution: (prototype): This is a solution promoted by Bouygues, composed of autonomous *car sensors*. Sensors are embedded in each parking spot and data is aggregated by a gateway called TOTEM. The gateway may be queried via FTP.

2) Charging stations: (prototype): These stations are used to charge electrical vehicles, they are manufactured by Schneider Electric. The charging stations are composed of *an RFID reader*: for users authentication and *two plugs* working independently. A station works autonomously, so that it manages the user authentication procedure, as well as the full process to charge a vehicle. Through a ModBusTCP [8] connection, it is possible to retrieve information such as the current state of the station (e.g a car is plugged, a car is charging, or a user has been identified) and the RFID values detected by the station. It is also possible to send remote commands, to reboot the station, for example.

3) Off-the-shelve products: : These equipment have been added to extend the scenarios with enhanced user experience:

- *Lights columns* : These columns have 4 lights (Blue, Red, Green, and Orange). They are placed at each parking spot to give feedback about the status of the related parking spot and/or charging station. They can be remotely controlled through an IP-Relay board.
- *Video camera*: Several IP-based cameras have been installed. It is possible to retrieve a video stream and for some of them to pilot their position.

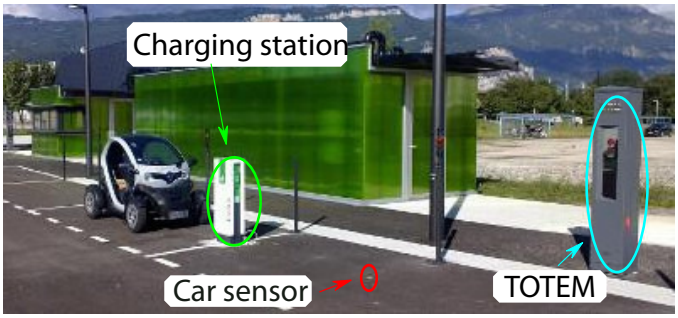

Figure 1. Picture of the parking

Figure 1 shows a picture of the parking. The blue circle highlights the sensors gateway (or TOTEM), the green circle highlights a charging station, and the red circle highlights one of the sensors embedded in a parking spot.

The prototype contains 3 charging stations with 2 plugs each, 5 instrumented parking spots (sensors and light columns), and 3 Video Cameras.

B. Scenarios

Three scenarios have been defined to target respectively the parking manager, the people responsible for the maintenance and the parking users. With LINC, the information are resources in bags, easily manipulated by rules to trigger actions given a specific context. Thus, from the same information (e.g. *a car is charged*) several actions may be taken: send an SMS to the car owner, store the information in a database, or raise an alert to the parking manager.

1) Parking maintenance scenario: This scenario focuses on the parking maintenance and administration. It allows to monitor and control all the equipment in the parking remotely, from a single interface. The interface offers a schematic view of the real parking with the current state of the equipment. This interface may be used for instance to display alerts caused by possibly faulty devices. The interface allows also to control manually some of the equipment such as lights or cameras independently from their usual behaviour.

2) Parking manager scenario: The parking manager is interested by statistics on parking spots and charging stations usage. The scenario provides an interface presenting the analysis of the usage of the charging stations and the parking spots. In this case, all the information from the equipment (i.e. sensor values, events from charging stations, etc) is logged in a database. Then, when the user requests it, statistics are computed and represented. Presenting these statistics in a unified manner will bring value to the parking manager. For instance, it may show that some users park on a charging station spot even though they never charge their car. Such information could be used to warn the parking user or increase the parking price when this situation happens. Another usage could be to detect cars parked for a long time, i.e. still connected to the charging station after the charge has ended. The manager can also learn from these statistics and adopt new strategies when planning future infrastructures. For instance, it is possible to determine which type of plugs are the most used.

3) User scenario: This scenario focuses on the user of the parking. The colour of the light column beside a parking spot is used to give information about the parking spot (e.g. busy or free) and the charging station (e.g. available, connected, or charging). The colour informs the user if his/her vehicle has been correctly connected, if it is charging, or if the charge has finished. Besides, this scenario also manages the authentication of users and the accounting for the parking usage. For instance, the identity of the user can be retrieved from the RFID reader; therefore, it is possible to associate the utilisation of a station with a user and an account. In addition, the user can be informed about the state of his/her account or charge through notification mechanisms such as SMS or e-mails.

4) Software components added to implement the scenarios:

- A *database* containing information about users (e.g. name, phone number), parking spots and charging stations usage (e.g. duration, tariff, user);
- *Simulators* for products not available yet. This allowed earlier integration of the products into the application in order to speed up the development process;
- *Notifications* to users and the parking manager.

V. Implementation

The case study raises several challenges:

- equipment were not all available at the beginning, or not with all their functionality;

- some of the equipment have an autonomous behaviour (they have not been designed to be integrated into a bigger product/system);
- high level information must be built by merging information of different equipment;
- time frame: this case study has been implemented in 3 months, with interfaces, equipment and scenarios evolving up to the last week.

This section presents how LINC has been used to face these challenges. In the proposed architecture, LINC has been used both for communication and coordination.

A. Software architecture

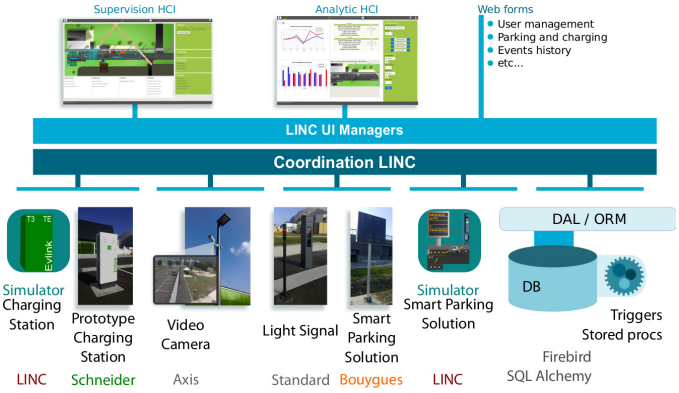

Figure 2. Software architecture

Figure 2 depicts the architecture. A simulator for the charging station has been developed as a Modbus slave whose registers values correspond to the expected values for each charging state; the same for RFID detection. Similarly, for the parking spot sensors, the behaviour of the TOTEM has been simulated thanks to sample files provided by the manufacturer. This allowed the integration of equipment still under development into scenarios before they were installed.

The database has been designed from scratch but it is now available as a ready to use framework that we already have used in another development in the field of connected health.

Most of the equipment integrated in the demonstrator are sensors and/or actuators or at least can be considered as such. We thus relied on the PUTUTU framework which already contained objects for some of the required technologies (e.g. Modbus used in the charging station, the IP based relay used for the column lights, and user notifications). For the missing parts, the templates coming with this framework reduced the development time and effort (e.g. the technology used for the car detectors has been added in a couple of days).

The coordination layer ensures, through LINC rules, the consistency of the system and implements the scenarios. The coordination rules define the core of the application, they decide which are the actions to take after the detection of an event (the presence of a resource in a bag).

B. LINC coordination rules

Several rules to implement the different scenarios were written. The rules are divided in four groups:

- 10 rules to *control the equipment*. They control, for example, the colour of the light columns according to the state of the associated spot and charging station;
- 12 rules to *manage charging cycles of the different stations* ;
- 9 rules to *log information into the database* for statistics purpose;
- 10 rules to *control the user interfaces*. They update the user interfaces according to values from equipment and they send events generated by the user to the different components of the application.

```
["App", "CCState"].rd(cc_id, st_id, badge, "charging")&
[st_id, "State"].rd("VEHICLE_CHARGED")&
["App", "UserInfo"].rd(badge, phone)&
::
{
 ["App", "CCState"].get(cc_id, st_id, badge, "charging");
 ["App", "CCState"].put(cc_id, st_id, badge, "charged");
 ["Alerts", "SendSMS"].put(phone, "Vehicle charged");
 ["DB", "Charge"].put(st_id, bagde);
}.
```

Listing 1. Rule to detect end of charge

Listing 1 shows an example of a simplified rule used to coordinate the charging cycle. The bag `CCState` refers to the state of the charging cycle associated to a charging station (`st_id`) and a user identity (`badge`). For each plug on a charging station, a unique charging cycle (identified by `cc_id`) can be active at a time. The precondition first asks the bag `CCState` of the `App` (Application) object for existing charging cycles. The second line waits for the resource modelling the end of charge event of the corresponding station. Finally the third line reads the user phone number. Then the performance part (after `::`) is triggered. Line 6 consumes the current state of the process (i.e. `charging`) and line 7 inserts the new state (i.e. `charged`). Then line 8 adds a resource in the `SendSMS` bag of the `Alerts` object to notify the user about the end of the charge. Finally the last line of the performance stores the end of charge event in the database (note that primary key information and time-stamp have been removed to make the rule more readable).

Regarding the application interfaces, they are developed on top of the UI Manager layer, and they rely on the coordination layer to interact with the physical equipment. The interfaces are described in the next section.

VI. User interfaces

This section presents the graphical interfaces developed for the first two scenarios. The *Live* interface allows to monitor and control the parking in real time. The *Statistics* interface displays statistics about the parking usage.

A. Live interface

Figure 3 shows the live interface provided by a `Layout` object with three frames (highlighted by a rectangle in the figure). The main frame (provided by an `MVC` object) shows the actual parking and the current state of the different equipment. This frame is an SVG file where each graphical element of interest can be managed through resources inserted in the bag `MVCStatus`.

Figure 3. Live Interface

```
["Parking","Sensors"].rd(spot, "Occupied") &
["LiveInt","map"].rd(c, spot) &
::
{
["Parking","Sensors"].rd(spot, "Occupied")
["LiveInt","MVCStatus"].put(c, "visibility", "visible") ;
["LiveInt","MVCRefresh"].put("frame1", "refresh") ;
}.
```

Listing 2. Rule to set car visibility attribute

Listing 2 shows the rule to show on the interface when a car is parked. The precondition consists of a `rd` on the bag `Sensors` of the `parking` object and a `rd` on the bag `map` of the `LiveInt` object. When a new resource is added in the bag `Sensors`, the rule reads the graphical entity associated to the spot and the performance is triggered. The performance checks that the sensor value did not change and updates the bags `MVCStatus` and `MVCRefresh` of the `LiveInt` object. This will automatically update the visibility attribute of the graphical entity in the live interface.

The live interface also shows further information about the parking spots, the charging stations, and the users currently using the stations. It triggers alerts to inform about abusive use of a charging station or faulty equipment. Then, the parking manager can interact with the different components. For instance, a click on a camera in the live interface opens the stream of the corresponding camera and a click on the red light of a column forces its state to red to indicate its unavailability to the parking customers.

```
["LiveInt","MVCControl"].rd(column_id, "click", colour) &
::
{
["LiveInt","MVCControl"].get(column_id, "click", colour);
["Column","command"].put(colum_id, colour) &
}.
```

Listing 3. Rule to force column colour

Listing 3 shows the rule to force the colour of a light column from the interface. The precondition contains one `rd` that triggers the performance when a resource indicating a click on a column is added. The performance consumes the resource and updates the model, here by sending the command with the correct colour to the object encapsulating the light column. Note that this change will then trigger a rule (similar to rule 2) that will update the MVCStatus and thus the interface.

These two examples of rules show how the physical world (sensors) triggers feedback to the user interface and how the user interface triggers actions on the physical world (actuators).

B. Statistics interface

Figure 4. Statistics Interface

Figure 4 shows the statics view of the application. The user can select some parameters for statistics computations and sends a new request to the statistics object (encapsulating the database). As soon as calculations are done, the interface is updated according to the new computed values. `Chart` objects are used to draw histograms, plots and pie charts.

The interface is composed of a configuration part (right column) and a display part. The right column configures the statistics computation:

- select the parking on which statistics are shown;
- display statistics on plugs or parking spot;
- period of monitoring (start/end dates, last day, week, month or year).

The display part is composed of four frames. The bottom right square displays the schematic view of the parking already seen in the live interface. However, even if it contains the same graphical elements, the context allows to use them for a different purpose. The user action is the same: click on a given graphical object. Information inserted in the bag is the same; however, the context, which is different, forces the execution of a different set of rules. This simplifies the re-usability.

If the user clicks on a plug or a parking spot statistics of the clicked element, for the selected period, are displayed. Finally the last three squares (top-left, bottom-left and top-right) display several statistics such as:

- ratio of occupied/free time per spot;
- ratio charging time/parking time;
- number of charges per hour, day, week, or month (according to the selected period).

VII. Related work

Several software approaches that provide rapid prototyping, targeting fast integration of heterogeneous equipment, have been proposed in the literature [9], [10], [11]. Similar to the work presented in this paper, these solutions propose the use of middlewares to facilitate the integration of new components and provide interactions between these components. However, most of these middleware solutions have been thought for a specific application domain lacking of flexibility. In addition, the encapsulation procedure being closely coupled with the application, reduces the possibility of code reuse.

Several coordination middlewares can be found in the literature, such as EgoSpace [12], LIME [13], TOTA [14] and Reo

[15]. As LINC, these middlewares rely on a resource approach, making space and time decoupling easier. However, when rapid prototyping is concerned, they do not offer development tools, ready to use frameworks, and user interfaces development framework.

Concerning the development of graphic interfaces, in most of the cases they are developed manually. Through the use of available tool-kits it is possible to use some predefined objects, such as buttons and menus, but most of the development is application dependent and cannot be reused. Mashups [16] are one example of approaches trying to integrate presentation components in a unique user interface. However, the lack of proper tools and models, increases complexity of Mashups requiring advanced programming skills. Yu et al. [17] propose a framework for graphic interfaces based on a very simple middleware; in this case the middleware allows the interaction between existing presentation components. LINC goes further by providing a method to define lightweight user interfaces with the same paradigm as the one used for the application logic itself. Thus, development of user interface is simpler, since once the graphic design has been done, it is only required to write the appropriate rules.

VIII. Conclusion

This paper has presented how LINC has been used to accelerate system prototyping. The prototypes targeted in this paper are applications that integrate several equipment from different vendors, with possibly some of the equipment still under development. Such prototypes are used to validate usages, demonstrate products at a fair or convince investors.

LINC provides a loosely coupled approach and a transactional rule engine. This characteristics permit to develop in parallel the integration of the different software and hardware components, the application logic and the user interfaces. High re-usability and simulation of not available components speed up the development. The transactional rule engine allows the programmers to focus on the application design without taking into account faulty equipment. This makes application easier to develop, faster to debug and more upgradable.

A case study of smart parking prototype including several equipment (some still under development), coming from different vendors, using different protocols, has been presented. The equipment were not designed to work together or even to be integrated in a more global system, since they have their own independent embedded behaviour.

This prototype has been developed in three months and demonstrated as part of a quite large event with several thousands of attendees. It is now used, on the one hand, for demonstration by the IRT Nanoelec and, on the other hand, as an open infrastructure in which other equipment can be integrated or other usages can be validated. We are currently working on the integration of a more elaborated lighting system and the validation of algorithms to adapt lighting, according to pedestrian detection.

This paper has also presented how rich user interfaces can be quickly built with LINC starting with standard designer tools such as Inkscape or Adobe Illustrator. Different interfaces have been provided for maintenance, parking exploitation, and customers on top of the same application objects and data.

This kind of fast development are of tremendous interest to reduce time to market, validate product during early stage, and anticipate commercial activities. Future work focuses on high level modelling and verifiable languages which can generate directly LINC rules. This will go one step further to decrease the prototype development time.

Acknowledge

This Work was supported by the French national program “Programme Investissements d’Avenir IRT Nanoelec” ANR-10-AIRT-05.

References

[1] M. Louvel and F. Pacull, “Linc: A compact yet powerful coordination environment,” in Coordination Models and Languages, ser. Lecture Notes in Computer Science, 2014, pp. 83–98.

[2] N. Carriero and D. Gelernter, “Linda in context,” Commun. ACM, vol. 32, 1989, pp. 444–458.

[3] T. Cooper and N. Wogrin, Rule-based Programming with OPS5. San Fransisco: Morgan Kaufmann, 1988, vol. 988.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency control and recovery in database systems. New York: Addison-wesley, 1987, vol. 370.

[5] G. E. Krasner and S. T. Pope, “A description of the model-view-controller user interface paradigm in the smalltalk-80 system,” Journal of Object Oriented Programming, vol. 1, 1988, pp. 26–49.

[6] F. Pacull et al., “Self-organisation for building automation systems: Middleware linc as an integration tool,” in IECON 2013-39th Annual Conference on IEEE Industrial Electronics Society. Vienna, Austria: IEEE, 2013, pp. 7726–7732.

[7] I. N. project, http://www.irtnanoelec.fr/en/.

[8] “ModBus Application Protocol Specification,” 2012. [Online]. Available: http://www.modbus.org

[9] R. Barraquand, D. Vaufreydaz, R. Emonet, A. Negre, and P. Reignier, “The omiscid 2.0 middleware: Usage and experiments in smart environments,” International Journal On Advances in Software, March 2012, pp. 231–243.

[10] T. Weis, M. Knoll, A. Ulbrich, G. Muhl, and A. Brandle, “Rapid prototyping for pervasive applications,” Pervasive Computing, IEEE, vol. 6, no. 2, April 2007, pp. 76–84.

[11] C. Côté, Y. Brosseau, D. Létourneau, C. Raïevsky, and F. Michaud, “Robotic software integration using marie,” International Journal of Advanced Robotic Systems, vol. 3, no. 1, March 2006, pp. 055–060.

[12] C. Julien and G.-C. Roman, “Egospaces: Facilitating rapid development of context-aware mobile applications,” Software Engineering, IEEE Transactions on, vol. 32, no. 5, 2006, pp. 281–298.

[13] A. L. Murphy, G. P. Picco, and G.-C. Roman, “Lime: A coordination model and middleware supporting mobility of hosts and agents,” ACM Transactions on Software Engineering and Methodology, vol. 15, no. 3, 2006, pp. 279–328.

[14] M. Mamei and F. Zambonelli, “Programming pervasive and mobile computing applications: The tota approach,” ACM Transactions on Software Engineering and Methodology, vol. 18, no. 4, 2009, p. 15.

[15] F. Arbab, “Reo: a channel-based coordination model for component composition,” Mathematical structures in computer science, vol. 14, no. 03, 2004, pp. 329–366.

[16] G. Di Lorenzo, H. Hacid, H.-y. Paik, and B. Benatallah, “Data integration in mashups,” SIGMOD Rec., 2009, pp. 59–66.

[17] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera, “A framework for rapid integration of presentation components,” in Proceedings of the 16th International Conference on World Wide Web, ser. WWW ’07. ACM, 2007, pp. 923–932.